

Toward Refactoring of DMARF and GIPSY Case Studies a Team 7 SOEN6471-S14 Project Report

Abdulrhman Albeladi	Ahmed Almessabi	Aber Abozkhar	Huda Mohamed	Jilson Thomas
Concordia University	Concordia University	Concordia University	Concordia University	Concordia University
Montreal, Canada	Montreal, Canada	Montreal, Canada	Montreal, Canada	Montreal, Canada
blady911@gmail.com	salmessabi@gmail.com	aberabozkhar2@gmail.com	h.m.haddar@gmail.com	jillztom@gmail.com

Zakaria Alomari
 Concordia University
 Montreal, Canada
 zakaria.alomari@gmail.com

Abstract—Software architecture is defined as the process of a well-structured solution that meets all of the technical and operational requirements, as well as improving the quality attributes of the system such as readability, Reliability, maintainability, and performance. It involves a series of design decisions that can have a considerable impact on the systems quality attributes, and on the overall success of the application. In this work, we start with analysis and investigation of two open source software (OSS) platforms DMARF and GIPSY, predominantly implemented in Java. Many research papers have been studied in order to gain more insights and clear background about their architectures, enhancement, evolution, challenges, and features. Subsequently, we extract and find their needs, high-level requirements, and architectural structures which lead to important design decisions and thus influence their quality attributes. Primarily, we reversed engineering each system's source code to reconstruct its domain model and class diagram model. We tried to achieve the traceability between requirements and other design artifacts to be consistent. Additionally, we conducted both manual and automated refactoring techniques to get rid of some existing code smells to end up with more readable and understandable code without affecting its observable behavior.

I. INTRODUCTION

The main goal of this work is to get better understanding and deep comprehension of the architecture of two case studies, Distributed Modular Audio Recognition Framework DMARF and General Intentional Programming System GIPSY, implemented in JAVA. To achieve this goal, we begin with studying different papers to analyse and investigate these two open source software (OSS), and to gain more insights and clear background about their architectures. We start by extracting and summarizing the core frameworks design artifacts for both OSSs such as; high-level requirements, fully dressed use cases, domain diagrams, and class diagrams. Next, we perform another step of identification of code smells. Some of these code smells are identified manually; others are identified automatically with the help of some tools such McCabe IQ, Logiscope, and JDeodorant. Subsequently, we apply different refactoring techniques on the existing code smells in order to get rid of these smells, and to restructure the code without changing its external behaviour, thus increase the readability,

understandability and reduce the complexity to make the code more maintainable as well as extensible. In addition, we come across implemented design patterns in both case studies, DMARF and GIPSY that deal with a specific problem in the design or implementation of software systems. Patterns can help us to include the existing well proven coding methods in software development which follow a good design methodology. Some of these patterns are Factory, Observer, Singleton, Adapter, Facade and many others. Lastly, we conduct JUnit test-cases to ensure that the applied refactoring techniques don't change the systems external behavior.

The paper is structured as follows: Section 1 presents two case studies DMARF and GIPSY predominantly written in Java with the goal of finding their needs, high-level requirements, and architectural structures. Also, it provides initial estimations of the size of both case studies. Section 2 defines the high level requirements, fully dressed use cases, and conceptual domain models. In addition, It shows how both systems can be merged and fused each other, where DMARF could use GIPSY's run-time for distributed computing instead of its communication technology. Section 3 presents UML class diagram for each system and the relationships between the classes in each diagram. Additionally, it shows how the traceability concept is achieved between requirements and other design artifacts to end up with a consistent design. List of the identified code smells and specific refactoring techniques will be used to include it in that section. Section 4 presents the implementation of refactoring and test cases as well as the conclusions of some insights gained from our experiments.

II. BACKGROUND

A. OSS Case Studies

1) *DMARF*: Distributed Modular Audio Recognition Framework (DMARF) is a distributed version of the original Modular Audio Recognition Framework (MARF). In order to illustrate DMARF, MARF first needs to be illustrated. As in [1] MARF is a Java open-source project for patterns recognition, signal processing, and natural language processing (NLP) [1]. MARF can run stand-alone, over the network,

or used as an application's library. There are several MARF's applications, like SpeakerIdentApp and FileTypeIdentApp [2]. MARF's architecture is a sequential pipeline with lake or even no concurrency when having a task of processing a bulk of voice samples. This problem has been resolved by extending the classical MARF to DMARF [3].

The pipeline stages or algorithms are designed in a modular way and provide extensibility feature to allow adding more algorithms. MARF contains pipeline stages that communicate with each other in order to obtain the required data. As shown in Figure 1 the pipeline consists of the four basic stages: sample loading, preprocessing, feature extraction, and training or classification [1]. In sample loading stage, the sample is first loaded and then converted to a supported type, for instance WAVE. Then, this sample is sent to preprocessing stage, where the sample gets normalized and filtered accordingly, so it can be prepared for feature extraction, which will be done in the next stage. After feature extraction comes training and classification, where the feature vectors could be considered as training data been stored or classified and the system learn from it [2].

The goal of extending classical MARF to DMARF as presented in [3] is to distribute the pipeline and make it runnable over a different groups of loosely coupled computers that work together closely. This flexibility of distribution these services aims to offload the bulk of multimedia or data to be processed to a higher performance servers that able to communicate while the collection of data can be done at several low-cost computers or PDAs without need to have processing and storage capacity, just they pass the collected data to the servers [3].

Yet another enhancement added to DMARF has been introduced in [4] by implementing disaster recovery, replication techniques and communication technology independence, such as RMI, CORBA, and Web Services (WS), which will allow the pipeline stages to communicate. WS makes the components even more interoperable and platform-independent, also, WS implementation in DMARF makes it even more widely available over the Internet.

DMARF is divided into layers, front-ends and back-ends application services, where all MARF's pipeline stages are placed in the front-ends, and the disaster recovery and replication techniques are implemented in the back-ends as shown in Figure 2. The front-end exists on the client side and on the server side. On the client side, a client application invokes services from the front-end on the server side, which means connect and query the servers. When a client application invokes services from the front-ends, the front-ends' services invoke neighbor services or the back-ends' services [2]. On the server side, the front-end services invoke other services from the back-end. At the same moment, the services like disaster recovery and service replication are a back-end for the client [4].

To achieve the management over MARF services [3], Simple Network Management Protocol SNMP is used to be integrated with use of common network devices and service

to provide the administrators with the capability to manage MARF nodes by using a familiar protocol. Moreover, it monitors and controls their performance, gathers statistics, and sets desired configuration. In contrast, DMARF components are stand-alone and distributed among different computers, and communicate to each other via RMI, XML-RPC, CORBA, and TCP connections, which leads to not understanding of SNMP Protocol. So each managed service in DMARF has to have:

- A proxy SNMP-aware agent for management tasks.
- A delegate instrumentation proxy to communicate with the specified service.

There are different points of view related to Distributed MARF: First, applying Autonomic Computing makes a system is self-managing autonomic system so that ongoing software and hardware complexity is decreased. Two properties of self-managing: self-healing and self-optimization are applied in DMARF, self-protecting autonomic property is also needed [1].

Second, a self-forensics considers as an autonomic property to boost the Autonomic System Specification Language (ASSL) framework of formal specification tools for autonomic systems to add the self-forensics autonomic property to enable generation of the Java-based Object-Oriented Intentional Programming language code linked with traces of Forensic Lucid for encoding contextual forensic evidence and other expressions. ASSL formal modeling, specification, and model checking has been applied to a number open-source, academic, and research software system specifications such as the Distributed Modular Audio Recognition Framework (DMARF) [5].

Third, Security can be breached by vicious attacks on the distributed systems such as DMARF, so some of the vicious attacks can be tackled by using middleware technologies while the rest can only be addressed by implementing robust security system. Thereby, JDSF security framework has been used as a security layer once communicating with nodes external to the local area network. JDFS resolves issues related to confidentiality, integrity, authentication, and availability. The proposed results of JDSF do blanket a wide cluster of the goals leaving malicious code detection unsolved [6].

The domain of DMARF system which is basically Audio and Voice Recognition Framework for patterns recognition, signal processing, and natural language processing (NLP) [2], [5]. Furthermore, DMARF extends to be used in different domains other than audio and voice processing such as speech recognition, forensics, security applications, text-independent Speaker-identification, language identification, natural language probabilistic parsing, and other classification applications. It also acts as a library in different applications or used as a source for learning and extension. Most of these applications are revolved around the multifaceted approach provided by DMARF. For example, one of DMARFs applications is SpeakerIdentAppt that has a database of speakers, where it can identify who people are regardless what they say. This application will extremely useful in law enforcement

agencies and police department for forensic analysis that need to identify speakers across all jurisdiction [3], [4].

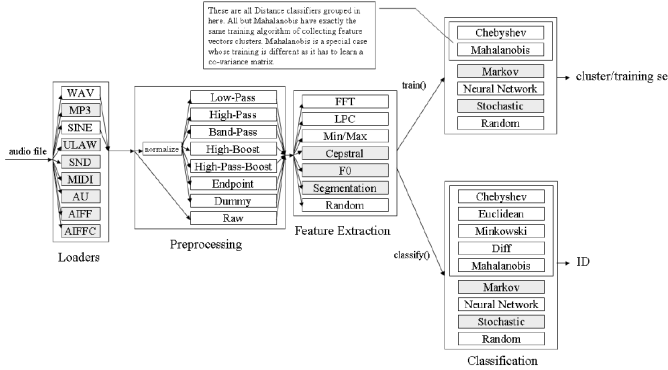


Fig. 1. MARF's Architecture [3, p.3]

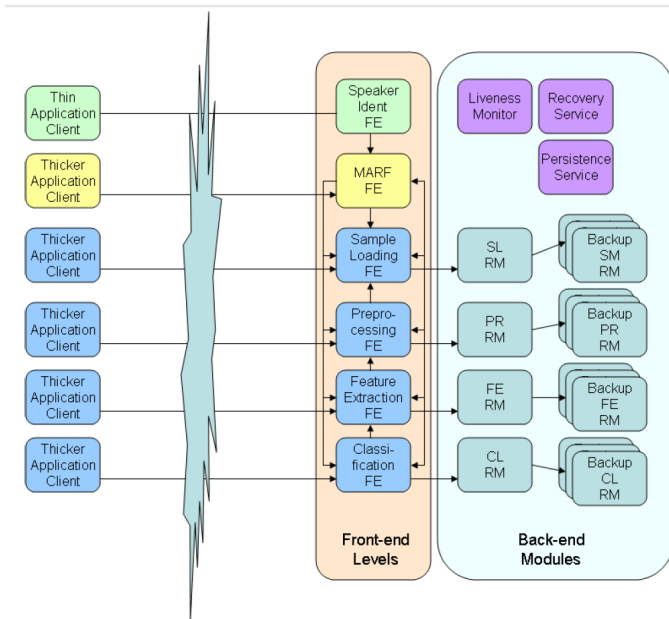


Fig. 2. DMARF's Architecture [3, p.4]

2) *GIPSY*: General Intentional Programming System (GIPSY) is a distributed demand-driven evaluation engine. The purpose of implementing GIPSY is to have a flexible compiler architecture that could read and compile multiple intentional programming languages [11]. Although the evolution of Intentional Programming IP has achieved a great maturity, there was a limitation in previous supported tools for this type of programming as well as lack of IP's visibility lead to the implementation of GIPSY system [9]. The main goal of GIPSY is to adapt to the fast development and the great diversity of the intentional family of programming languages. Furthermore, through hybrid programming in combination with standard procedural languages, it allows reusing of legacy code, and enables the use of different distributed execution middleware at run-time. The system provides these characteristics easily,

in a way that makes the users can create new Intentional Programming Languages, use of different procedural languages, or middleware technologies [8].

GIPSY's architecture is a multi-tier architecture and have the advantage of using modular development for the compiler components. It is composed of three main components, namely General Intensional Programming Compiler (GIPC), General Education Engine (GEE) and Intensional Run-time Programming Environment (RIPE). Each component has its own architecture or design which means independent and can be replaced or maintained without affecting the other components. In addition, the subsystems are designed towards generality, flexibility and efficiency [9].

The GIPC component, is the main compiler for GIPSY, and in charge of translating IPs Program to C, which in turn will be compiled in a standard way. The GEEs architecture is a distributed multi-tier architecture, which allows one or more instances for each tier. This architecture is similar to peer-to-peer architecture, which makes GIPSY more robust when a failure occurs in any trier or node. GEE component is mainly concern about generating tasks that can be executed in a parallel way using the computation model of demand-driven model. The GEER is language independent and a run-time resources dictionary compiled from GIPL program. GEERs instances are created by GIPC when it compiles a program [11], [7]. Figure 3, Shows GIPSY Architecture.

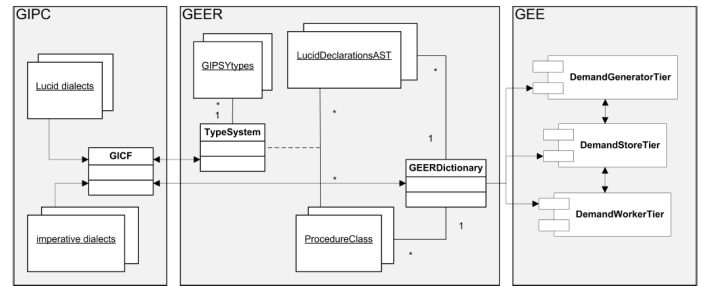


Fig. 3. The GIPSY Architecture [2, p. 148]

GIPSY has shown an effectiveness in meeting its main purposes mentioned above, as well as it is evolved in a flexible way. An example of that, GIPSY provide context-oriented multidimensional reasoning of intensional expressions without limiting the scope of evaluation of math expressions as GIPSY is based on the higher-order intensional logic (HOIL). HOIL combines functional programming with different intensional logics to allow explicit context expressions to be evaluated in GIPSY [11].

GIPSY solves the distributed architecture problems of the generic runtime system. Previously, the architecture was not fully integrated and detailed working flow had to be modified [10]. The proposed solution for the above mentioned problems are based on the GLUs generator-worker architecture. Through multiple design iterations, it extended to be a multi-tier architecture and applied the high level designs using wrapper classes for each newly introduced tier type, mainly DWT

(Demand Worker Tier), DGT(Demand Generator Tier), DST (Demand Store Tier), and GMT(General Manager Tier). This Solution is an evolution of the GIPSY runtime systems original architecture.

Another aspect that GIPSY consider is investigating hybrid programming language in local and distributive way. Therefore, a Demand Migration Framework (DMF) is designed and introduced in the execution environment of GIPSY in order to provide a generic and dynamic infrastructure for the purpose of distributed demand-driven communication [7]. DMF was built using two different java middleware technologies, Jini (Java-based and service-oriented middleware technology) and JMS (Java-based and message-oriented middleware technology). However, the two middlewares are separated from GIPSY and from each other. In order to make the GEE scalable enough to compute the heavy hybrid programs in a distributive way, there was a necessity to unify the two java middleware technologies, Jini and JMS. This has been resolved by doing refactoring. First, by making JMS and Jini part of the same interchangeable framework's implementation. Redefine the role of demand dispatcher and Transport Agent and Compare Jini and JMS with JVM, compare their APIs and compare the ease of deployment and startup [7].

Based on the conceptual level, implementing the self-forensics within the Autonomic System Specification Language (ASSL) and General Intensional Programming System (GIPSY) is described through their founding core works. ASSL formal modeling, specification, and model checking has been applied to a number of open-source, academic, and research software system specifications, such as GIPSY. ASSL is a framework that provides a multi-tier specification model for Autonomic System (AS) as well as ASSL allows expressing ASs, as a set of interacting Autonomic Elements (AEs), at three main levels: AS level, AS Interaction Protocol (ASIP) level, and AE level [12].

GIPSY has illustrated the effectiveness of intentional programming to solve critical problems in various domains such as tensor programming, distributed operating systems, and software versioning, hybrid programming language, demand driven educative, and forensic.

3) *Summary:* MARF services uses the implementation of three components such as RMI, COBRA and Web services (WS). But, RMI and COBRA has got some problems such as need of RemoteException throws during stubs creation and generation of data structures for COBRA. For a greater portability using HTTP service for the MARF, this WS is implemented for the DMARFs COBRA and RMI. This WS can even communicate with these two DMARF components even through plain UDP or TCP protocols. An algorithm with ASSL of the pipeline pattern recognition of the DMARF is used to create a ADMARF self-protection specification model which will be very much functional in autonomous environments. However, it is not a complete autonomous ADMARF specification model. For the minimal testing, generation of proxy agent and compilation is enough in debug form of MIBs. To allow self-forensics property implementations in the ASSL

toolset, some preliminary work is done. To protect the security aspects of many uncontrolled public networks, a solution of JDSF is proposed and it covers a wide range of security aspects for scientific research and experimental distribution systems.

The multi-tier components will offer a high scalability and flexibility to the GIPSY at runtime when it is implemented and fully tested. Some new packages were developed and defined for this multi-tier implementation. The design of GIPSY architecture allows components to be replaced at runtime or at compiler-time so as to improve flexibility and scalability and allows automated generation of hot spots for support of programing language. In the area of flexible hybrid intentional research programing, the GIPSY is well off for the scientists and researchers. In the GIPSY programming paradigms, the central value is the concept of context as a first class which explores the lucid programing language family. The distributed transport of the demands DMS has implementations in Jini, plain RMI, and JMS and the POC integration of the middleware is implemented using Jini and JMS. In the future, the other components in the GIPSY can be improved to work with unison by refactoring and cleaning up the code.

The methodology used to analyze both DMARF and GIPSY projects source code was simple by using a tool called SonarQube [13]. SonarQube (previously known as Sonar which mean is the central place to manage code quality, offering visual reporting on and across projects and enabling to replay the past to follow metrics evolution [14]) is an open web-based application platform to manage code quality. We used SonarQube 4.3.2 Released which fixes migration issues that can occur on specific cases like Migrations that convert technical debt from hours to minutes are too slow[15]. Also we used Sonar Runner v2.4 This version provides various improvements and bug fixes[15].

Sonar Server has been used in order to interpret and analyze both projects considering Apache HTTP Server to be associated with Sonar Server. Afterwards, each project has used sonar-project-properties file to set sonar properties. Thereby, the team has corporated to run and access Sonar Runner by command line as well as Sonar Runner reads the selected project files and collected required measurements. As a result, the values is presented in a browser. In order to get theses measurements, Sonar did not compile niether systems. Also, the measurements were only on Java files.

The required measurements are presented in Table I as follows:

Table I , it demonstrates several measurements for two different open-source systems DMARF, and GIPSY respectively. After executing SonarQube, it shows DMARF system exceeds slightly less the half of files and classes number than GIPSY. Likewise, DMARF in number of packages is greater by three times than GIPSY system. In terms of number of methods, both systems seem quite close to each other considering DMARF is little bit bigger whereas line of text in GIPSY considers less than DMARF. Moreover, number of statements and comments lines appear away bigger in DMARF

TABLE I
DMARF AND GIPSY MEASUREMENTS

Measurements	DMARF	GIPSY
Files	948	608
Packages	377	164
Classes	978	666
Methods	6760	6276
Line of Text	123261	140828
Number of statements	28478	56078
Comment Lines	19939	13814
Line of code	73393	104199

than the other whereas line of code of DMARF is less than GIPSY. Thus we conclude that DMARF is more complex as compared to GIPSY because DMARF's number of files, classes and methods are higher.

III. REQUIREMENTS AND DESIGN SPECIFICATIONS

A. Personas, Actors, and Stakeholders

1) DMARF:

Persona:

Ahmad Aljohani is 35 years old. Ahmad is a male adult, and have a good experience with computers and the Internet. He Received Masters degree in Software Engineering from Concordia University. Ahmad now is working as a developer at CIT company, and have attended many workshops and conferences in how to develop well designed software systems. Also, he has a good knowledge in developing and integrating software systems, and has develop couple software systems. He is currently interested in developing a survey software system, that would allow its users to create surveys and distribute them. This software will have an option to allow its users to get authenticated through voice. This kind of feature would needs to identify its users through voice recognition and get them authenticated when there is a match. He does not have enough time to implement such a feature, instead he needs a good library system that can identify users via voice, and match them to the existing users to get them authenticated. In addition, he needs that library to be simple to use and get benefit from it. He never liked to use a library that does not have a good documentation or user manual. In addition, It would create a big problem for Ahmad if the system did not identify accurately the right users for the right accounts. Performance is an essential quality for Ahmad's software system because if it is not fast enough, his users would not use his system. Ahmad get frustrated easlly when he uses complex systems' interface. Moreover, he never liked to ask his colleagues to teach him how to use a system because he thinks it would make him look stupid. Ahmad wants his system to be portable, so it can run in different platforms.

Actors:

Writer Identification Application processes tasks of scanned hand-written documents to identify the writer, such as

students' exams verification and personal checks identification. This application can achieve its goal by using MARF's approach to define a common set of integrated APIs for the pattern recognition pipeline [16]. This technique is useful for biometric modality with applications in forensic and historic document analysis [17]. Writer Identification Application is considered as a primary actor for MARF.

JDSF allows to work with several types of data storage in order to anticipate different level of security algothrims and methodologies in certain environment. JDSF covers several aspects to keep data privacy, to integrate and to authenticate data stemming from a trusted source for DMARF. Once an attack takes place somehow, JDSF shall protect DMARF's data and secured them. Besides, It does capture the configuration data related to setting connection or any other properties related to the distributed systems [6]. Therefore, JDSF is considered as a secondary actor.

Text-Independend Speaker Identification Application processes amount of voice samples to test MARF's functionalities so this application tells who, gender, accent, and spoken language of the speaker using MARF's pipeline. Thereby, Text-Independend Speaker Identification application contributes to share services, to arrange as a modular, and to facilitate adding new algorithms for use or experiments. JSDF uses these algorithms implemented to recognize patterns and to process tasks. Not only does that, but also it is leveraged from MARF's backbone particularly pipeline to get data needed [18]. Thus we considered this application as a primary actor.

Communication technology is a standard architecture for a distributed system, and it is designed to allow distributed systems like GIPSY and DMARF to interoperate in a heterogenous environment, where systems can be implemented in different programming languages and/or deployed on different platforms. It is based around the concept of a client application using the services available on a remote machine, or server. The object's interface represents a contract between the client and the server. This interface is written as a Java interface for Java RMI, in IDL for CORBA, and in WSDL for web services [21]. Therefore, communication technology can be considered as a secondary actor for DMARF and GIPSY.

2) GIPSY:

Persona:

Anton is a 34 year old Manipal University professor and a freelance software developer. He has a Phd in the field of software engineering and has many experience in software versioning and hybrid, tensor programmings. He has got an industrial experience of 7 years as the lead/senior developer for a famous software developing company and very good knowledge in Java, C, C++ and a limited knowledge in .NET. Currently he is trying to develope a fingerprint recognition application for the Police Department. He wants a platform which is independent of the programing language to compile and execution since he might use different languages to develop the system. Therefore, there should be a flexible

compiler for the system. It should also be compatible to solve any security issues that may arise in the fingerprint recognition environment. Also, he is experiencing some problems with the distributed architecture systems that he is working on as part of the project. Currently, he is looking for a system that could encode an image sample into a Forensic lucid language, and then get compiled by GIPSY to investigate the fingerprint. The developing system should be made up of different components which will further interact to each other. Therefore, he is mainly interested in a modular development for the system components.

Actors:

Software Versioning system merge all versions that have a similar version description into a single version, such as Lemur system using an intensional versioning technique. Lemur system uses intensional programming implemented in a demand-driven computation framework. It can achieve its goal by using GIPSY techniques [9], [11]. Software Versioning system is considered as a primary actor for GIPSY.

Hybrid programming: is an integration of intensional and imperative languages that is required for different needs such as reusing legacy code. GIPSY provides developers with a platform where they can create new Intensional Programming Language IPL, enables the use of different procedural languages, or middleware technologies in an easy way with minimal technical knowledge by increasing the automation of the extension process [19]. The developers are considered as primary actor.

Finger Print Identification App is an application that receives an image sample from the target system, like in our situation Crime Investigation system. Then, it encodes that image into Forensic Lucid language. Later, GIPSY receives the Forensic Lucid code. Then, GIPSY compiles and evaluates it. The Finger Print Identification App is considered as primary actor.

B. Use Cases

1) DMARF:

Use case name: Identify Speaker's Voice

Scope: System under investigation.

Level: User level

Primary Actor: Survey System.

Stakeholders and Interests: Survey System:

- Wants the **SpeakerIdentApp** to identify the user accurately.

Preconditions:

- 1) **Survey system** has a **voice sample** ready to be uploaded.
- 2) Both **Survey system** and **SpeakerIdentApp** are connected to the Internet.

Postconditions:

- 1) A **user** gets identified.
- 2) A reply with the matching result.

Main success scenario:

- 1) **Survey system** wants to identify a **user**.
- 2) **SpeakerIdentApp** asks to upload the **voice sample**.
- 3) **Survey system** upload the **voice sample**.
- 4) **SpeakerIdentApp** processes the **voice sample**.
- 5) **SpeakerIdentApp** matches it with registered users in **Survey system**.
- 6) **SpeakerIdentApp** replies to **Survey system** with the matching result.

Extensions:

The system shall crash at any time, the user needs to close and reopen it.

1a: Network connection fail:

1a.1: before uploading voice file:

- 1) system user should look for good enough connection.
- 2) Go to a step 1.

1a.2 After uploading voice file:

- 1) Go to step 2

3a: Survey system fails to upload the sample.

- 1) **SpeakerIdentApp** asks to upload the voice sample again.
- 2) Go to step 3.

4a: **SpeakerIdentApp** fails to process the voice sample due to Network Failure:

- 1) **SpeakerIdentApp** looks again for a new or former connection.
- 2) Go to step 2.

5a: **SpeakerIdentApp** fails to match the voice sample with any registered users.

- 1) **SpeakerIdentApp** reply with an error message to survey system that it failed to identify any user.
- 2) **SpeakerIdentApp** asks to upload the voice sample again.
- 3) Go to step 3.

Special requirements:

- Voice sample must be clear.
- Voice sample must be uploaded within 15 second.
- Confirmation of **SpeakerIdentApp** (or reason for failure) to be provided to the Survey system within 20 seconds of submission.

Technology and data variations list:

- The Survey system should be able to upload arbitrary type of voice sample.

Frequency of occurrence:

- Could be nearly continuous.

Miscellaneous:

- What if the Survey system could not be able to upload the voice sample?
- What if the SpeakerIdentApp did not identify the right user for the right account?
- What if SpeakerIdentApp does not support all list of voice extensions?

2) GIPSY:

Use case name: Identify Fingerprint

Scope: System under investigation.

Level: User level

Primary Actor: Crime Investigation System

Stakeholders and Interests: Crime Investigation System:

- Wants to identify the fingerprint image accurately.

Preconditions:

- 1) **Fingerprint image** is **scanned** and ready to upload.
- 2) Both **Crime Investigation system** and **Fingerprint Identification system** are connected to the Internet.

Postconditions:

- 1) A suspect has been identified from his/her fingerprint.
- 2) A result of matching or not has been sent to the police system.

Main success scenario:

- 1) The **Crime Investigation system** wants to identify a suspects fingerprint.
- 2) The **Fingerprint Identification system** asks to upload the **image**.
- 3) **Crime Investigation system** uploads the **image** to the **Fingerprint Identification system**.
- 4) **Fingerprint Identification system** processes the **image**.
- 5) **Fingerprint Identification system** compares the uploaded **image** with the registered suspects.
- 6) **Fingerprint Identification system** replies with the matching result.

Extensions:

3a. The Crime Investigation system failed to upload the image.

- 1) Fingerprint Identification system displays an error message.
- 2) Fingerprint Identification system asks to upload the image again.
- 3) Go to step 3.

4a. If the Fingerprint Identification system fails to process the image because of insufficient network connection.

- 1) Fingerprint Identification system finds an appropriate network connection.
- 2) Go to step 3.

5a. If the Fingerprint Identification system does not recognize the fingerprint image.

- 1) Fingerprint Identification system responds with a message that no suspect was found.
- 2) Go to step 2.

Special requirements:

- The image should be clear enough to be processed
- Sufficient time for upload the image should not be more than 20 seconds.
- The system should be secure enough to prevent any incorrect usage.
- Confirmation response should not exceed more than 20 seconds.
- Robust recovery for the system in case of failure

Technology and data variations list:

- The system allows various image format readings

Frequency of occurrence:

- Might occur frequently.

Miscellaneous:

- What if a problem happened while uploading a fingerprint image?
- What if the image format is not supported by the Fingerprint Identification system?

C. Domain Model UML Diagrams

1) **DMARF**: Figure 4, Shows the DMARF's domain model with **SpeakerIdentApp**. The **Survey system** can be used for any type of survey related applications. It needs the **end users** to get authenticated by their voices to use the system. A **SpeakerIdentApp** is used for identifying the end users from their recorded **voice samples**. This identification will be done by using the sound recognition in **MARF**. First, the users will be asked to record their voice on login process in the **Survey system**. This **voice sample** will be recorded by the end user through the **Survey system** and uploads to **DMARF** through the **SpeakerIdentApp**. It is then moved to **MARF** to compare it with the already existing voice records.

The **MARF** pipeline is used to compare and to process the pattern recognition and to send the feedback back to the **SpeakerIdentApp** and then to the **Survey system**. If the **MARF** gives a positive reply back to the **SpeakerIdentApp**, and then to the **Survey system**, the user will be authenticated and be given permission to access the **Survey system**. If the

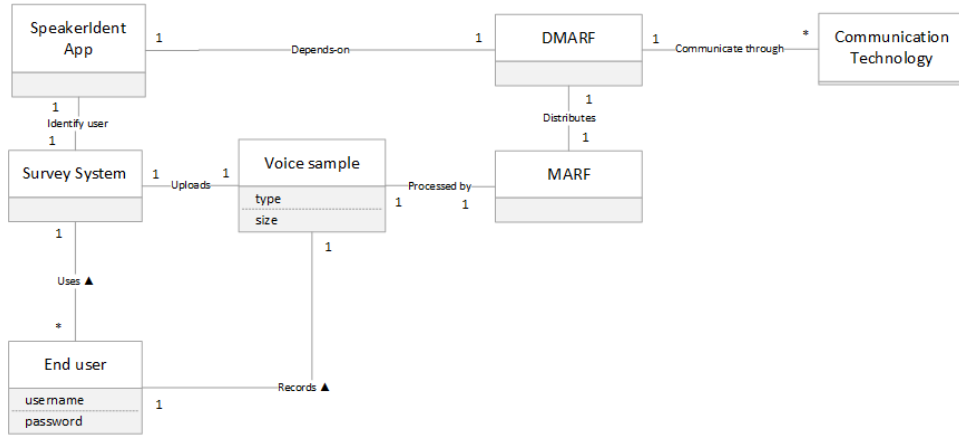


Fig. 4. DMARF's Domain Model

MARF gives a negative feedback or if the patterns do not match, then the **Survey system** will not authenticate that a particular user from accessing the system. Only one voice sample can be given to the **Survey system** at a time by one user. The **Survey system** can be used by many users at the same time.

2) **GIPSY**: Figure 5, Shows the **GIPSY**'s domain model. Using a **Fingerprint Identification application** a **crime investigation system** can identify the **fingerprints** of a suspicious person who already has a record. The end users will be the **police officers** who are investigating any crimes. Many **policemen** can use the **crime investigation system** to scan and upload different **fingerprints** at the same time to check a suspicious **fingerprint**. **Crime Investigation System** will demand **Fingerprint Identification application** to identify the recent image. The **Fingerprint Identification application** will assign an image of **fingerprint** to **Forensic Lucid expression**, which already has been encoded the image (evidence) in the consistent syntax. In **GIPSY**, **GIPC** will be responsible for parsing such **Forensic Lucid specification** and **GEE** will be responsible for executing it. By executing the **Forensic Lucid**, it will match the encoded evidences/images with the new encoded image of the suspicious **fingerprint**. An evaluation of whether it matched or not will be sent to the **Police officer**. **GIPSY** requires some **communication technology** services such as **RMI**, **WS** for communication.

3) **Fused DMARF-Over-GIPSY Run-time Architecture (Do-GRTA)**: Figure 6, Shows the fused **DMARF-Over-GIPSY** domain model. For **DMARF** to be distributed over **GIPSY**, discarding communication technology protocols, such as **CORBA**, **RIM** and **WS**, is needed and replacing them with **GIPSY**'s tiers. Basically, **GIPSY** is a Multi-Tier Architecture which is fully demand driven where the demands are generated by the tiers and migrated to other tiers using the Demand Store by Transport Agent **TA** [7], [11].

Basically, to get the advantages of the demand driven architecture that **GIPSY** has, each node of the pipeline has two components, demand generator component and a demand worker component. Generally, the generator is the one who

generate the demand. Starting with the sample loader pipeline, the application will create the sample loader pending demand. The demand generator receives the demand, and store it in the demand store. The sample loader demand worker picks the sample loader pending demand up and carry out the functional computations requested to sample loader stage. Later, the sample loader demand worker generates preprocessing demand. Consequently, the preprocessing demand generator will receive the preprocessing demand and store it in the demand store. The preprocessing demand worker picks up pending or unprocessed preprocessing demand, then it executes the preprocessing stage. After that, the feature extraction demand will be generated by the preprocessing demand worker. The feature extraction demand will be received by the feature extraction generator and stores it in the demand store. The feature extraction worker picks up the feature extraction pending demand and executes the feature extraction stage. Then, the feature extraction worker generates the classification pending demand. The classification generator receives the classification demand and stores it in the demand store. Then, the classification worker picks up the classification pending demand and executes the classification stage. After that, the classification worker produces a result of the classification stage. The semantic of the **MARF**'s pipelines is maintained by distributing it on a multi-tier architecture like **GIPSY**. Thus, its scalability is improved [7], [11].

D. Actual Architecture UML Diagrams

1) **DMARF**: The entire design of **DMARF** class diagram is summarized in eight main modules and their relationship as follows: **MARF**, **SpeakerIdentApp**, **WSUtils**, **RMIUtils**, **ISampleLoader**, **Sample**, **Ipreprocessing**, **IFeatureExtraction**, and **IClassification**. Indeed, **MARF** uses communication technology type of interfaces, **WSUtils** or **RMIUtils**, to pick up either manually or automatically which communication technologies is supposed to be in the system design. These interfaces are defined in **marf.net** and they are used in reflection instantiation utils. Next, **RMI**, **WS** server and client interfaces are branched in the hierarchy. They used to set and get in-house-made

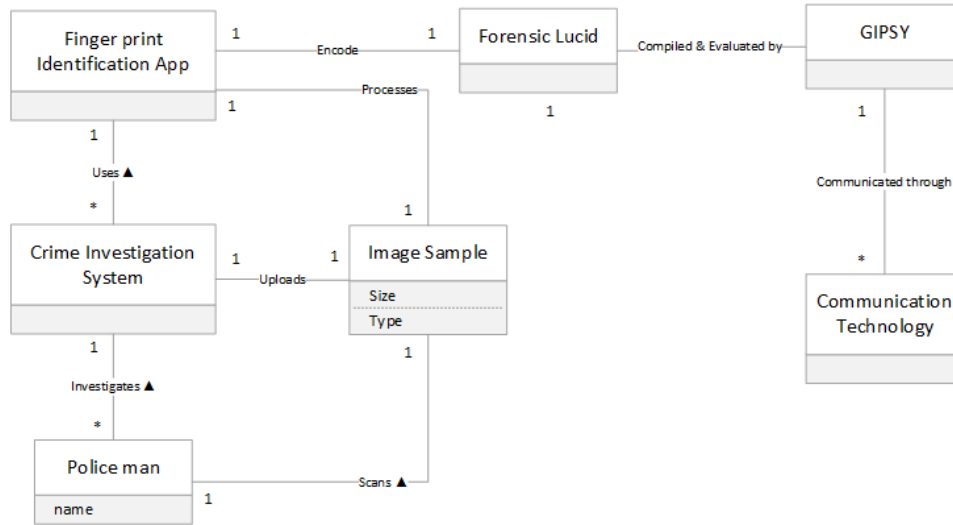


Fig. 5. GIPSY's Domain Model

RemoteObjectReference which isn't a true object reference as in RMI, yet it encapsulates the necessary service location information. These communication technologies are associated somehow with WAL logging and transaction recovery. Also, there are some monitoring modules designed as well.

Then, SpeakerIdentApp recognizes MARF and commences recognition pipeline. SpeakerIdentApp has several important methods assisting MARF to let sample be identified. Some methods are getVersion, getConfigString, and SetDefaultConfig. Besides, MARF loads a sample file into IsampleLoader which requires concrete preprocessing to process the sample file after IsampleLoader will ask Sample Class to upload it. Preprocessing will get back to Sample in array list after normalizing has been done. Afterwards, MARF processes this file sample in Preprocessing, and generates preprocessed sample. Likewise, MARF does some further processing with IFeatureExtraction in order to get the sample.

Furthermore, MARF associates together with IFeatureExtraction to extract features and to generate features vector. Features which have been generated need to be classified, so MARF and IClassification will address this issue properly. After classifying extracted features has been done, FeatureExtraction get results of array, and determine the feature ID. Next, it sends this result into Result. MARF manages this result in both SpeakerIdentApp and Classification then MARF gets this result ID into Result.

In Preprocessing Class diagram, its summarized into several components (modules) associating with main one, Sample, which are IPreprocessing, IPreprocessingRMI, FFTFilter, IFilter, and Preprocessing. Sample needs Preprocessing to processes and manages samples using FFTFilter to classify and to preprocess these samples. In IPreprocessingRMI, it initiates communication technology needed to process this sample as well as to normalize it.

Moreover, Sample Loader class diagram gets leveraged from MP3Loader to load sample file as well as to write audio

data, to read audio data, and to save a sample. SampleLoader has loadSample, saveSample, getSampleSize, getSample, setSample, updateSample, and other methods assisting Sample to get audio format, to set audio format, to get next chunk, to reset array mark, to clone, and to get sample array. In a word, DMARF class diagram stems from DMARF's domain model with some solution problem models which express how MARF address and process its samples in an order and management to have good results. Not only does MARF identifies a user sample, but also it classifies, preprocesses it and extracts features. Communication technologies such as RMI, WS have been used for communication with WAL, managed his serialized WAL entity handle by storage (Storage Manager) and transaction, interacts with delegate type. The StorageManager class provides implementation of serialization of classes in binary as well as compressed binary formats. Not only that, but It also has facilities to plug-in other storage or output formats. In terms of Database, all result set, and classifications states which is written in SpeakerIdentApp are stored in database [2]. Figures 7, 8, 9, 10 and 11, show the DMARF' class diagrams.

DMARF's domain model (Conceptual architecture) covers most world problems. Components such as MARF communicate with samples through the communication technology to distribute services. Once a sample has been uploaded in Survey System by the end-user, SpeakerIdentApp will identify the user depending on DMARF. A user could record his voice. In contrast, the actual system architectures shows the internal components and the interactions between them. For instance, in the former a voice has represented in Sound Sample whereas in the latter is represented as Sample.

From our conceptual classes, most of the conceptual classes maps to the actual classes in DMARF class Diagram, and they are as follow:

- Speaker Ident App maps to SpeakerIdentApp
- Voice Sample maps to Sample

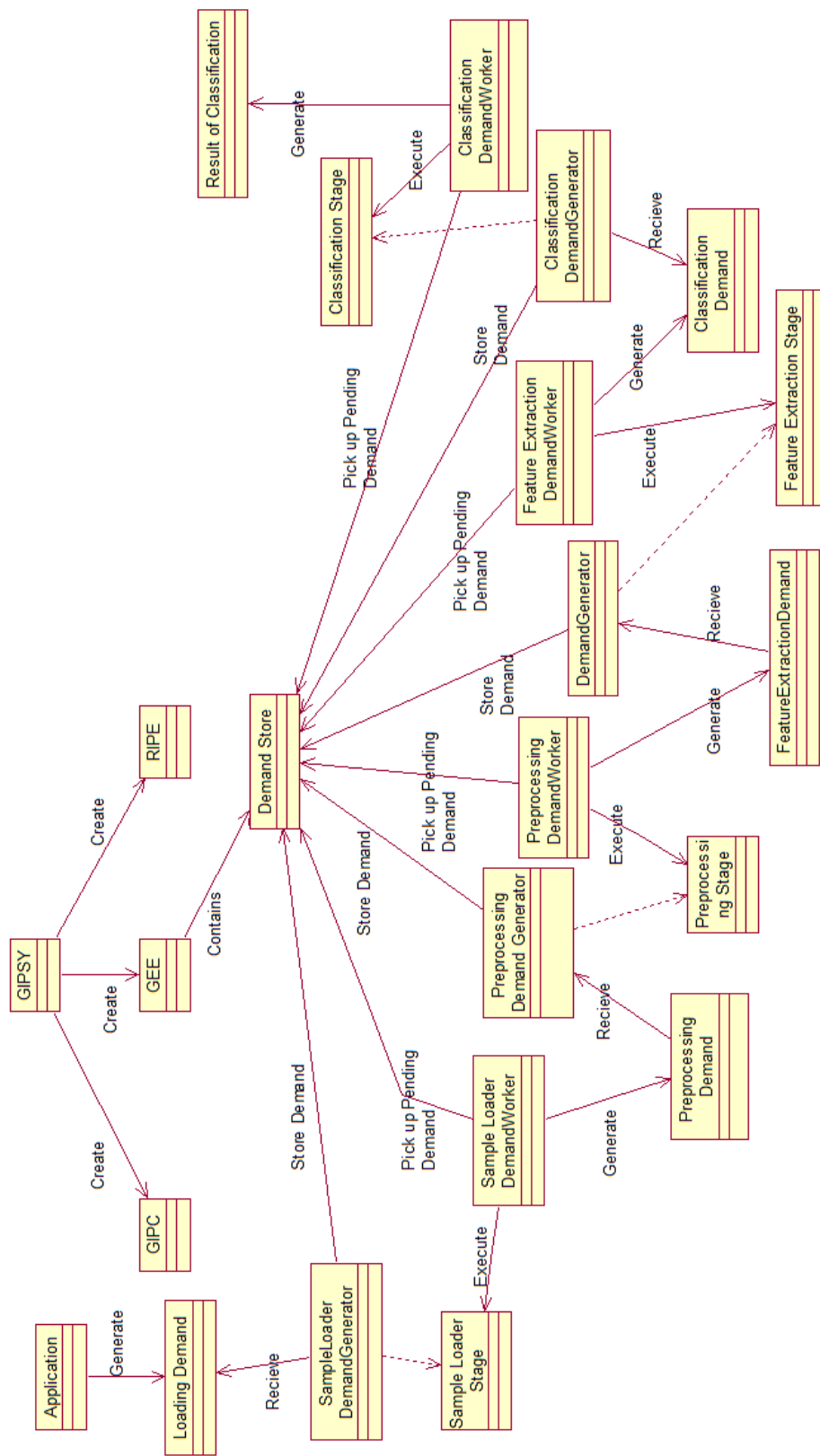


Fig. 6. Fused DMARF-Over-GIPSY Domain Model

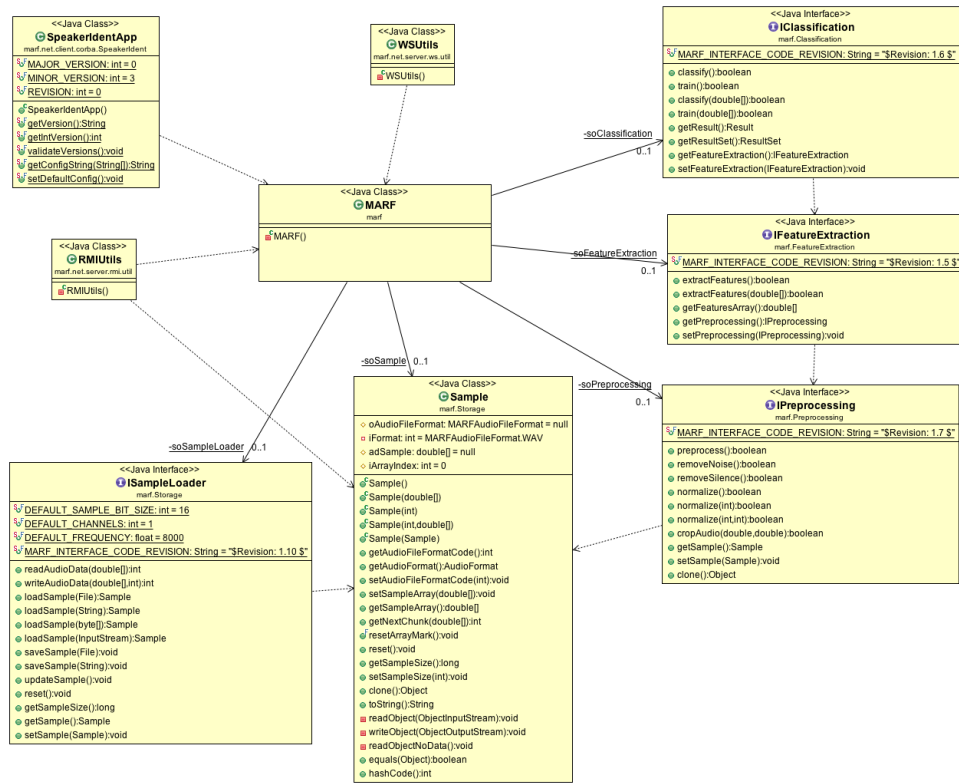


Fig. 7. DMARF's Class Diagram

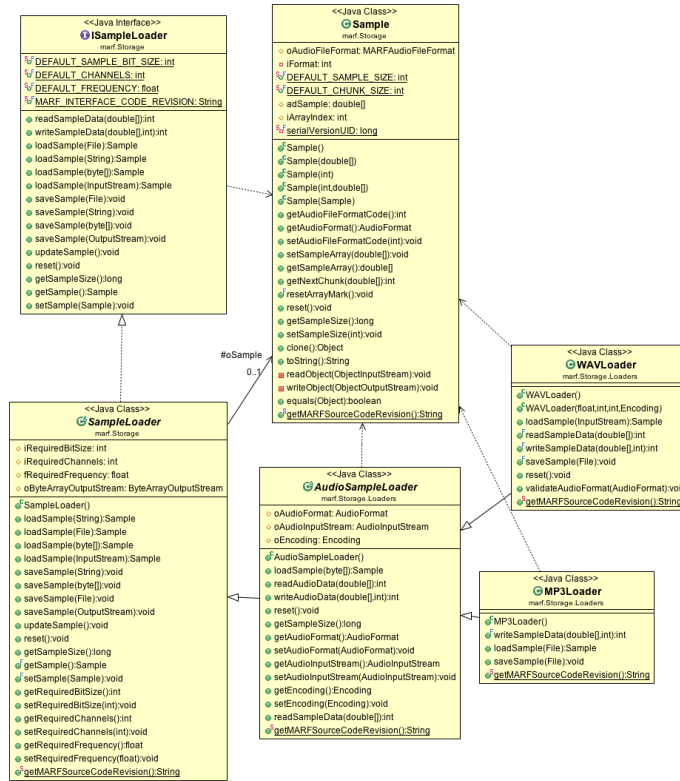


Fig. 8. DMARF's SampleLoader Class Diagram

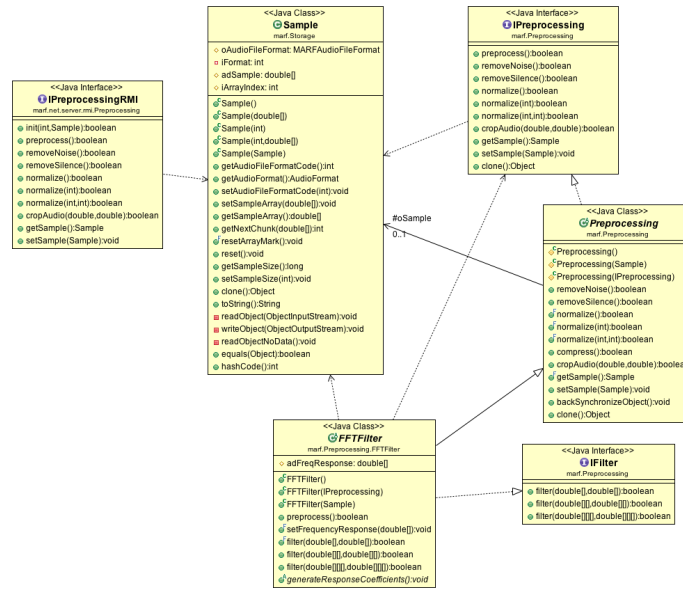


Fig. 9. DMARF's Preprocessing Class Diagram

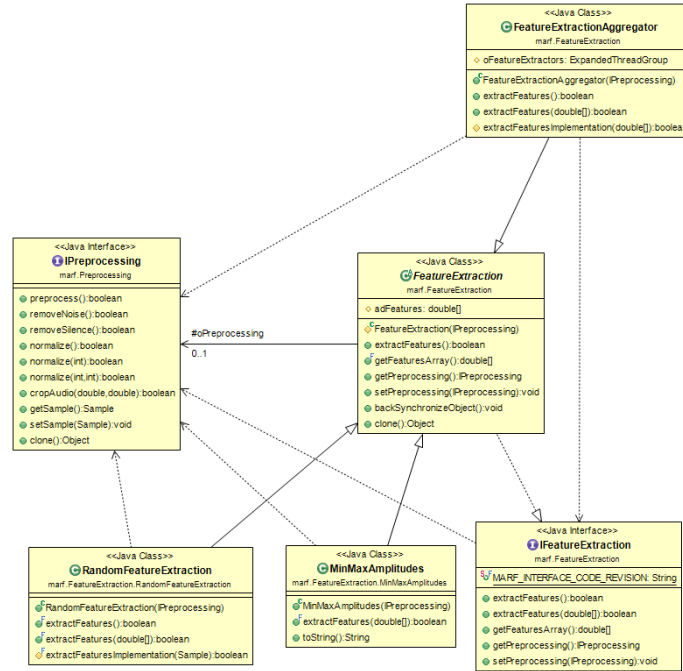


Fig. 10. DMARF's Feature Extraction Class Diagram

- DMARF maps to DMARF
- MARF maps to MARF's SampleLoader, Preprocessing, FeatureExtraction, Classification

Conceptual classes uses classes from the actual system architecture. Since DMARF deals with pattern recognition, the conceptual classes uses the pattern recognition and its components to authenticate the user in its domain problem. Conceptual classes such as SpeakerIdentApp, SoundSample, etc uses corresponding similar classes from the actual classes. This means, the conceptual system is fully dependent on the actual

DMARF architecture and its components. In the conceptual architecture, the voice sample uploads to DMARF through the SpeakerIdentApp and DMARF is communicated through the communication technology. Therefore, the conceptual diagram is more of an application level architecture of DMARF. While the DMARF actual architecture explains about the components and interaction between the components.

We have used ObjectAid UML tool to help us build the class diagram for both DMARF and GIPSY. ObjectAid UML is a plugin for Eclipse that is used to reconstruct the class

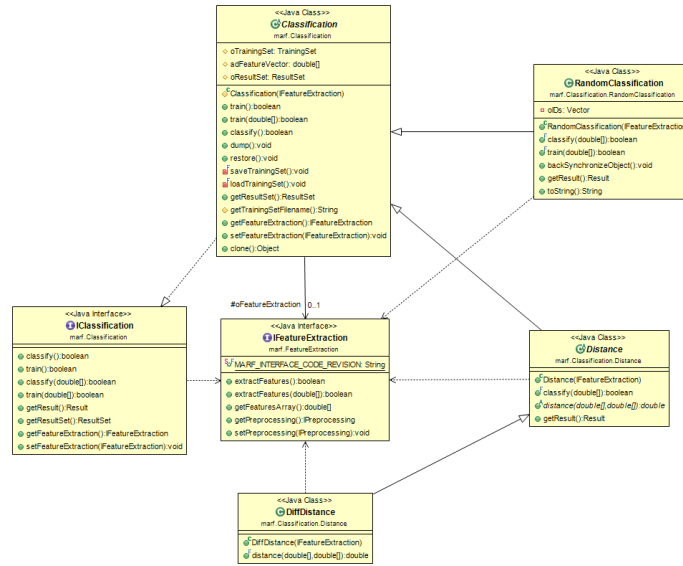


Fig. 11. DMARF's Classification Class Diagram

diagrams for the actual system source code. This plugin allows to reverse engineer the class diagrams just by drag and drop of the required java classes from the source code. Also, it allows to show the associations and dependencies between different classes. In addition, we have the option to hide/show different types of attributes and methods.

2) *GIPSY*: The whole design of *GIPSY* class diagram contains major modules and some their relationships between these modules: *GIPSYProgram*, *GIPC*, *GEE*, *RIPE* and other important components. Several design patterns have been applied such as observer, and singleton pattern. The singleton is applied to ensure one controller is created for each tier on each node, and further factory method pattern lets the subclasses to specify the tier objects it controls. We abbreviate unimportant classes; we focus only the interested classes which relative into our domain model. *GIPSYProgram* has dictionary, communication procedure, and *GEERSignature* associate with other main components *GEE* and *GIPC*. *GEE* class is integrated to invoke multi-tier services via the main entry of the engine and to accommodate missing development. *GEE*'s controller will adjust and process services over a particular tier. To let *GEE* start, *GEE* uses created API.

Next, *GEE* employs factory method to instantiate desired tier type of arbitrary instances, and then it starts or stops them directed using API. *GEE* delegates some activities to node controller and proceeds to the program execution if there is one to execute after having a tier get started. As execution processes, demands are generated and handed into the tier that responsible for the delivery and results computation to get back with warehouse store so that it caches principles.

GIPC has Preprocessor, Dictionary, Translator, and others that cooperate together to investigate Intentional Programming Language, requiring *GEE*, *RIPE* Controller, DFG editor, and Abstract Syntax Tree in *GIPSYProgram*. *GIPC* uses *GEER-Generator* to link *GIPCProgeam* so Abstract Syntax Tree could

```
public abstract class SampleLoader implements ISampleLoader{
    protected Sample oSample = null;

    public Sample loadSample(final String pstrFilename) throws StorageException{
        return loadSample(new File(pstrFilename));
    }

    public Sample loadSample(byte[] patFileData) throws StorageException{
        return loadSample(new BufferedInputStream(new ByteArrayInputStream(patFileData)));
    }

    public void saveSample(final String pstrFilename) throws StorageException{
        saveSample(new File(pstrFilename));
    }

    public void updateSample() throws StorageException{
        double[] adSampleArray = new double[(int)getSampleSize()];
        readSampleData(adSampleArray);
        this.oSample.setSampleArray(adSampleArray);
    }

    public long getSampleSize() throws StorageException{
        return this.oSample == null ? 0 : this.oSample.getSampleSize();
    }

    public final Sample getSample(){
        return this.oSample;
    }

    public final void setSample(Sample poSample){
        this.oSample = poSample;
    }
}

public class Sample implements Serializable, Cloneable{
    public Sample(double[] padData){
        try{
            setAudioFileFormatCode(MARFAudioFormat.WAV);
            setSampleArray(padData);
        }
        catch (InvalidSampleFormatException e){
            throw new RuntimeException(e);
        }
    }

    public Sample(final int piFormat) throws InvalidSampleFormatException{
        setAudioFileFormatCode(piFormat);
    }

    public Sample(final int piFormat, double[] padData) throws InvalidSampleFormatException{
        setAudioFileFormatCode(piFormat);
        setSampleArray(padData);
    }
}
```

Fig. 12. DMARF's class diagram Classes and the Relationship Source Code First Part.

clone, dump, and sow tree. *GIPC* has compiler to parse, compile, and get abstract syntax tree. *ForensicLucidCompiler* is one type of *GIPC* compiler has *ForencisLucidParser*.

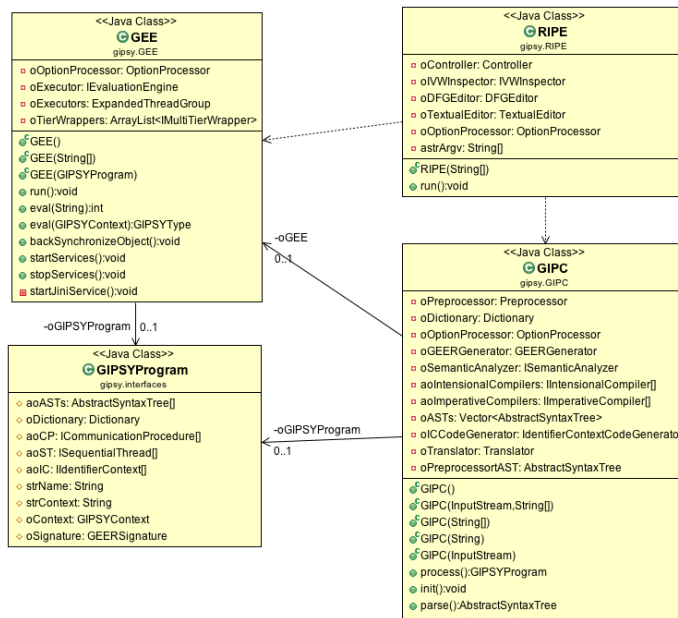


Fig. 14. GIPSY's Class Diagram

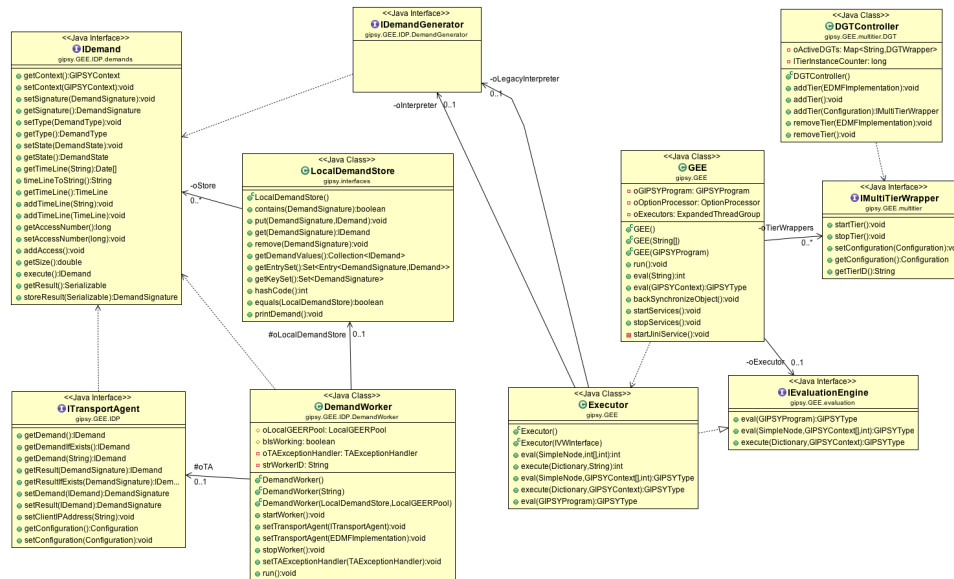


Fig. 15. GIPSY's GEE Class Diagram

In addition, GEE has major components: DemandGenerator, LocalDemandStore, DemandWorker, and TransportAgent. LocalDemandStore is considered as an observer for its subjects, DemandGenerator which gets or sets state, and gets or sets access for DemandWorker object who interests to know about DemandGenerator new status which informing the concrete subject, TransportAgent, to return subject state [7].

The conceptual architecture is an application of the the actual system architecture of GIPSY. That is, the conceptual model uses the GIPSY architecture to function certain tasks. In the conceptual architecture, GIPSY is used to compile the FingerPrint Application and it is communicated through

the communication technology. Therefore, the conceptual diagram is more of an application level architecture of GIPSY. Whereas, the GIPSY actual architecture explains about the components and interaction between the components. Figures 14, 15, 16, and 17, show the GIPSY' class diagrams.

From our conceptual classes there are only two classes that maps to actual classes in GIPSY class Diagram, and they are as follow:

- GIPSY maps to GIPSYProgram, GIPC, GEE, and RIPE.
- Communication technology maps to RMI, and CORBA.

The actual classes obtained from the background reading and the actual architecture of the system are closely asso-

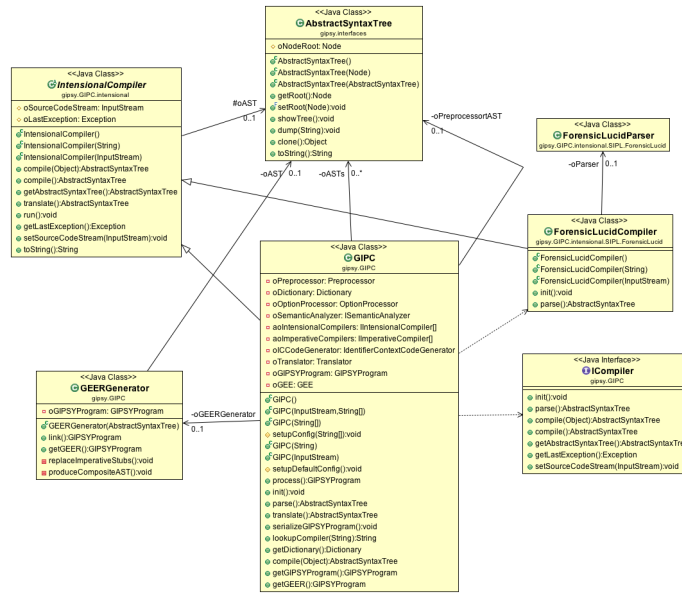


Fig. 16. GIPSY's GIPC First Class Diagram

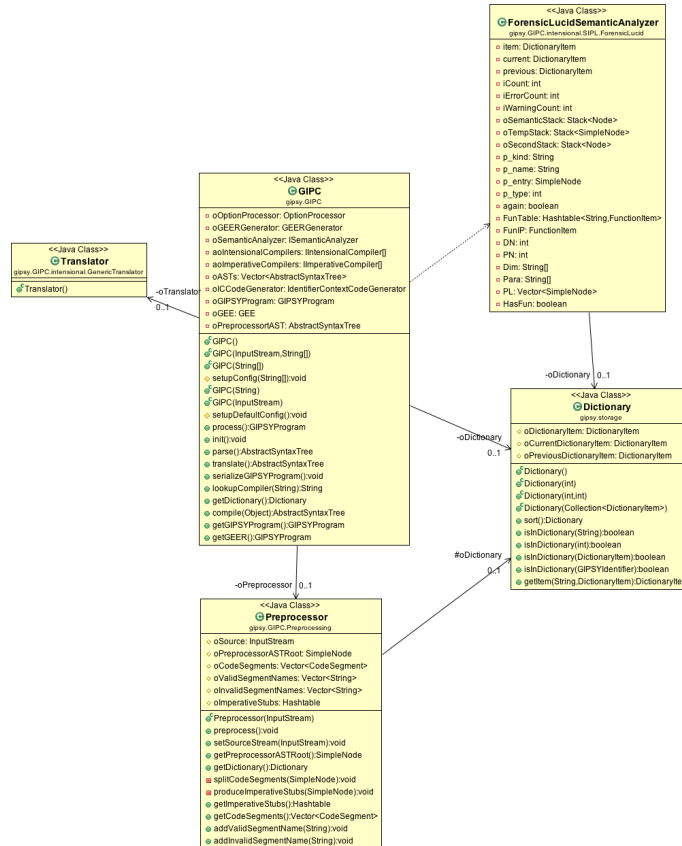


Fig. 17. GIPSY's GIPC Second Class Diagram

ciated with the conceptual classes. Some conceptual classes which comes in the application level apart from the GIPSY implementation has some discrepancy between them, whereas the conceptual classes for the GIPSY has similar actual

classes. That means, the actual architecture of the system has similarity with the conceptual architecture when it comes to the interaction between the GIPSY's components.


```

public Sample(final Sample poSample) throws InvalidSampleFormatException{
    Sample oCopy = (Sample)poSample.clone();
    setAudioFileFormatCode(oCopy.iFormat);
    setSampleArray(oCopy.adSample);
}

public synchronized int getAudioFileFormatCode(){
    return this.iFormat;
}

public synchronized AudioFormat getAudioFormat(){
    return this.oAudioFileFormat.getFormat();
}

public synchronized int getNextChunk(double[] padChunkArray){
    int iCount = 0;
    long lSampleSize = getSampleSize();
    while(iCount < padChunkArray.length && this.iArrayIndex < lSampleSize){
        padChunkArray[iCount] = this.adSample[this.iArrayIndex];
        iCount++;
        this.iArrayIndex++;
    }
    return iCount;
}

public synchronized long getSampleSize(){
    return this.adSample == null ? 0 : this.adSample.length;
}
}

```

Fig. 13. DMARF's class diagram Classes and the Relationship Source Code Second Part.

IV. METHODOLOGY

V. REFACTORING

A. Identification of Code Smells and System Level Refactorings

1) **DMARF**: After inspecting the DMARF codes, we come across with the following code smells:

In the **MARF** class, we found three code smells namely code duplication, Long method and complex if statements, and they are as follow:

- train () and train (Sample poSample) - Code duplication: In both the methods, some codes are repeated, which causes code duplication. In this case, we could extract the duplicated codes from both the methods and make it as a method, so that it can be called from both these methods.
- startRecognitionPipeline(Sample poSample) - Long method: This particular code smell is due to large chunk of codes in the specified method which makes it hard to understand. To avoid the code smell, we could extract the codes from this long method and create another methods and call it as needed.
- startRecognitionPipeline(Sample poSample) - Complex if statement: This method contains complex if statements and it may lead to longer processing time and logical errors. So the possible solution is to simplify the if conditions.

In the classes **Distance** and **RandomClassification**, one code duplication code smell is found. getResult() - Code duplication:

- Both the Distance and RandomClassification classes have the same superclass Classification class. Both classes have the getResult() method with similar behavior. This means that the method is duplicated content and to solve this problem is to pull up this method to the superclass and inherit its behavior.

In the classes NeuralNetwork, one code duplication code smell is found. generate() - Code duplication:

- In this method, some codes are repeated, which makes code duplication. Therefore, we could extract the codes from this method and make different methods and call it from the original method.

Another code smell found among the following classes is speculative generality. It is affected by the classes AIFFCLoader, AIFFLoader, AUloader, MIDILoader, MP3Loader, SNDLoader and ULAWLoader. These classes are not implemented but just put it as empty. The possible way to avoid is to remove these Loader classes.

2) **GIPSY**: Based on the analysis made on the GIPSY package, we found the following code smells:

In **GEE** class, three methods are affected by the code smell long method:

- GEE(String[] argv) - Long method: This method has long codes which are prone to logical errors or complex structure. So the possible solution is to extract the codes and make corresponding methods and call them whenever wanted.
- startServices() - Long method: Similar to the above method, it also has long codes which are prone to logical errors or complex structure. So the possible solution is to extract the codes and make corresponding methods and call them whenever wanted.
- eval(GIPSYContext poContext) - Long method: This method also has long codes which are prone to logical errors or complex structure. So the possible solution is to extract the codes and make corresponding methods and call them whenever wanted.

Similarly In the GIPC class, we found one possible code smells, which are: GIPSYProgram process() - Long method:

- This method is very long, open to errors and has complex code structure that makes it hard to read and understand. So the possible solution is to extract fragment of codes and make them separated methods and call it as needed.

In the JavaCompiler class, we found two code smells:

- init() and parse() - Code duplication: These two methods are inherited from the ImperativeCompiler class and has similar codes. So the possible solution is to extract the codes from both these classes and make a single method in the super class and call it as needed.

As we found in the DMARF, we also found some Speculative Generality code smells in the GIPSY package as well. The following classes (PerlCompiler, CCompiler, CPPCompiler, FortranCompiler, PythonCompiler) are not fully implemented. So we could remove them from the code to avoid any possible confusion or errors.

B. Specific Refactorings that Will Be Implemented in PM4

1) **DMARF**: We identified the common design problems or code smells in DMARF's source code and their corresponding refactoring techniques among the methods, and they are as follows:

- checkSettings() Method from MARF class - Complex if statement
- startRecognitionPipeline(Sample poSample) Method from MARF class - Long method
- generate() Method from NeuralNetwork class - Code duplication
- startRecognitionPipeline(Sample poSample) Method from MARF class - Complex if statement

For DMARF, there is no test cases for NeuralNetwork, Distance Class, RandomClassification Class and Classification class. Therefore, in this case we can write the JUnit tests manually. Eclipse supports the creation of JUnit tests via a wizard. Naming the new JUnit tests would be TestNeuralNetwork, TestDistance, TestRandomClassification and TestClassification, like the TestWaveLoader test case. We will write these tests to make sure that the code behaviour did not change after the refactoring. For MARF class, there is test case called test under default package.

2) **GIPSY**: For the refactoring in GIPSY, we will try to refactor the code smells in the classes GEE, GIPC and JavaCompiler. For the class GEE, we will try to refactor the methods GEE(String[] argv) and startServices() has long method code smell. In addition, in GIPC class, we encounter a long method code smell for the method GIPSYProgram process() and we will try to refactor it too. Regarding to the class JavaCompiler, we will refactor the code duplication code smells for the methods init() and parse() by pulling them up to ImperativeCompiler.

For the testing in GIPSY, we have the gipsy.tests.Reggression class to apply the test cases to see if the refactoring makes any bad changes to the output of the system. This test class is mainly useful for the GEE and GIPC class code smell refactorings. There is no test cases for JavaCompilerclass in GIPSY's source code, but if we may to create JUnit test, it would be TestJavaCompilerclass, like the TestForensicLucidSemantic test case. These test classes will help us to identify if the refactoring has affect the behaviour of that class or not. If the outcome of the test cases is similar before and after each refactoring, then we could say that the refactoring was effective and did not affect the outcome of the system. If the outcome is different from the outcome before refactoring, then we could conclude that the refactoring badly affected the system and should make a revoke of the doings and give more attention to the refactoring.

C. Identification of Design Patterns

In order to understand and identify design patterns, we have used ObjectAid UML Explorer tool. ObjectAid UML Explorer is optimized for the quick and easy creation of UML class from existing Java source code and libraries. It uses the UML

notation to show a graphical representation of existing code. ObjectAid UML plugin for Eclipse is used to find out or reverse engineer the interacting classes from the actual classes. Based on the Design-Pattern Identification, the corresponding interacting classes are identified using this eclipse plugin [27].

```
public class MARF
{
    private static ISampleLoader soSampleLoader = null;
    private static final void startRecognitionPipeline()
        throws MARFException
    {
        soSampleLoader = SampleLoaderFactory.create(siSampleFormat);
    }
}

public final class SampleLoaderFactory
{
    public static final ISampleLoader create(final int piSampleFormat)
        throws InvalidSampleFormatException
    {
        ISampleLoader oSampleLoader = null;
        Debug.debug("Requested loader: " + piSampleFormat);

        switch(piSampleFormat)
        {
            case MARF.WAV:
                oSampleLoader = new WAVLoader();
                break;
            case MARF.MP3:
                oSampleLoader = new MP3Loader();
                break;
            case MARF.ULAW:
                oSampleLoader = new ULAWLoader();
                break;
            case MARF.AIFF:
                oSampleLoader = new AIFFLoader();
                break;
            case MARF.AIFFC:
                oSampleLoader = new AIFFCLoader();
                break;
            case MARF.CUSTOM:
            {
                try
                {
                    oSampleLoader =
                        (ISampleLoader)MARF.getSampleLoaderPluginClass().newInstance();
                }
                catch(Exception e)
                {
                    throw new InvalidSampleFormatException(e.getMessage(), e);
                }
                break;
            }
            case MARF.TEXT:
                oSampleLoader = new TextLoader();
                break;
        }
        return oSampleLoader;
    }
}
```

Fig. 19. DMARF's Factory Pattern Code

1) **DMARF**:

Factory Pattern:

The factory pattern is a widespread design pattern. This pattern comes under creational pattern, and it helps to create an object in the best way by providing varies way to create it. This pattern comes in handy when there is a need to create an object without directly specifying the type of the object at compile time, and allows the client to select the desired class at runtime to create the object. To implement the factory pattern, an interface class has to be created and be the only place where these objects can be instantiated. Client classes

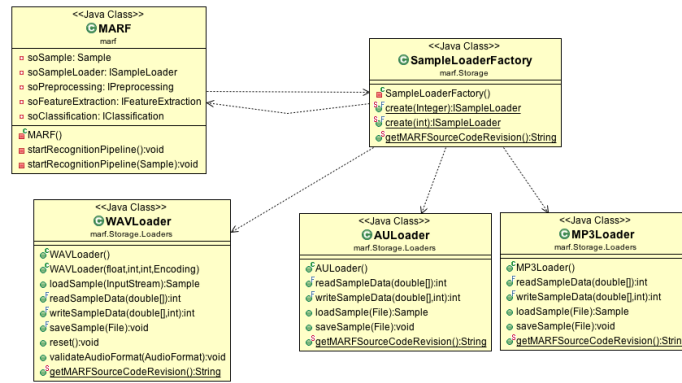


Fig. 18. DMARF's classes involved in the Factory pattern.

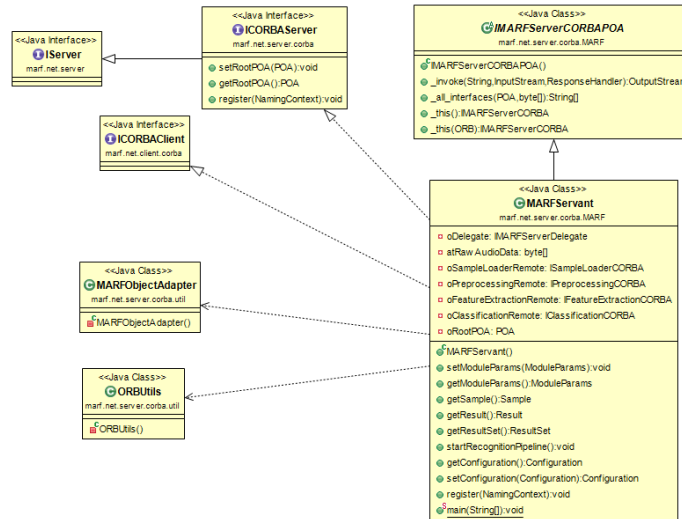


Fig. 20. DMARF's classes involved in the Adapter pattern.

select the type of object they need, and send it to the factory class. Then, depending on the sent type of the needed object, the factory class creates the object, and returns it back to the client class [22].

Factory pattern is used in DMARF's sampleloader pipeline stage to support loading different voice extensions, like WAVE, MP3 and many more. A SampleLoaderFactory class is implemented to allow MARF class to select the type of the sample to be loaded by SampleLoader class. Then, it instantiate the selected type of object and returns it back to MARF class to load that sample. Figure 18, shows DMARF's classes involved in the factory pattern. Figure 19, shows source code for factory pattern in DMARF.

Sometimes, creating objects is a complex process, and if it were not solved probably, it could cause lots problems in the code. Especially, when the needed object is unknown before run time. Code duplication would have been used to solve this problem. However, it is not the right way to do it. The best way for this kind of problem is to use the factory pattern [22].

Adapter Pattern:

The adapter design patterns replicates the plug adapter, in that it converts the current systems interface into another interface by wrapping the entire interface and creating the desired new interface. In the other words Adapter design pattern is one of the structural design pattern and its used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an Adapter. The benefits of this pattern are that it allows objects to be encapsulated by a new class structure and creates new interfaces that match the class that invokes it [23]. There are two approaches whereas implementing Adapter pattern : class adapter which mean uses java inheritance and extends the source interface, and object adapter which mean uses Java Composition and adapter contains the source object , however both these approaches produce same result [24].

In DMARF, the adapter pattern is applied while implementing the CORBA services, a data type adapter had to be made to adapt certain data structures that came from MARF.idl to the common storage data structures. Thus, the MARFObjectAdapter class was provided to adapt these data structured back and forth with the generic delegate when

```

public class MARFServant
extends IMARFServerCORBAPOA
implements ICORBAServer, ICORBAClient
{
    private IMARFServerDelegate oDelegate = null;
    private byte[] atRawAudioData = null;
    private ISampleLoaderCORBA oSampleLoaderRemote = null;
    private IPreprocessingCORBA oPreprocessingRemote = null;
    private IFeatureExtractionCORBA oFeatureExtractionRemote = null;
    private ICClassificationCORBA oClassificationRemote = null;
    private POA oRootPOA = null;

    public MARFServant()
    throws Exception
    {
        super();
        new Logger("marf.server.corba.log");
        this.oDelegate = new BasicCMARFDelegate();
    }

    public void setModuleParams(ModuleParams poModuleParams)
    {
        this.oDelegate.setModuleParams(MARFObjectAdapter.getMARFModuleParams(poModuleParams));
    }

    public ModuleParams getModuleParams()
    {
        return MARFObjectAdapter.getCORBAModuleParams(this.oDelegate.getModuleParams());
    }

    public Sample getSample()
    {
        return MARFObjectAdapter.getCORBASample(this.oDelegate.getSample());
    }

    public Result getResult()
    {
        return MARFObjectAdapter.getCORBAResult(this.oDelegate.getResult());
    }

    public ResultSet getResultSet()
    {
        return MARFObjectAdapter.getCORBAResultSet(this.oDelegate.getResultSet());
    }

    public Configuration getConfiguration()
    {
        return MARFObjectAdapter.getCORBAConfiguration(this.oDelegate.getConfiguration());
    }

    public Configuration setConfiguration(Configuration poConfiguration)
    throws CORBACommunicationException
    {
        try
        {
            return MARFObjectAdapter.getCORBAConfiguration(this.oDelegate.setConfiguration(MARFObjectAdapter.getMARFConfiguration(poConfiguration)));
        }
        catch (MARFException e)
        {
            throw MARFObjectAdapter.getCORBACommunicationException(new CommunicationException(e));
        }
    }
}

```

Fig. 21. DMARF's Adapter Pattern Code First Part

```

public class MARFObjectAdapter
{
    private MARFObjectAdapter()
    {
    }

    public static final CommunicationException
    getMARFCommunicationException(CORBACommunicationException poCORBAException)
    {
        return new CommunicationException(poCORBAException);
    }

    public static final CORBACommunicationException
    getCORBACommunicationException(CommunicationException poGeneralException)
    {
        return new CORBACommunicationException(poGeneralException.getMessage());
    }
}

public interface ICORBAServer
extends IServer
{
    void setRootPOA(POA poPOA);
    POA getRootPOA();
    void register(NamingContext poNamingContext)
    throws Exception;
}

```

Fig. 22. DMARF's Adapter Pattern Code Second Part

needed. In the other words, MARFObjectAdapter class is responsible for translating common MARF data structures to CORBA and vice versa. Figure 20, shows a UML class diagram for the adapter pattern in DMARF. Figure 21 and 22, shows source code for adapter pattern in DMARF.

Singleton Pattern:

Singleton pattern is one of the simplest design patterns that addressed these concerns: create only one instance of a class,

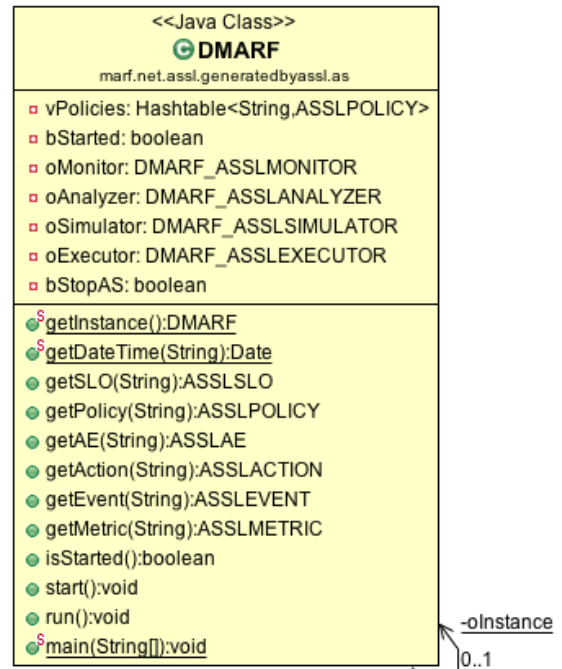


Fig. 23. DMARF's classes involved in the Singleton pattern.

```

public class DMARF extends Thread
{
    static private DMARF oInstance = null;

    private DMARF ( )
    {
    }

    ASSLO.put("PERFORMANCE",marf.net.assl.generatedbyassl.as.dmarf.asslo.PERFORMANCE.getInstance());

    vPolicies.put("SELF_HEALING",marf.net.assl.generatedbyassl.as.dmarf.assself_management.SELF_HEALING.getInstance());

    ACTIONS.put("STARTSELFHEALING",marf.net.assl.generatedbyassl.as.dmarf.actions.STARTSELFHEALING.getInstance());

    EVENTS.put("LOWPERFORMANCEDETECTED",marf.net.assl.generatedbyassl.as.dmarf.events.LOWPERFORMANCEDETECTED.getInstance());

    EVENTS.put("PERFORMANCENORMALIZED",marf.net.assl.generatedbyassl.as.dmarf.events.PERFORMANCENORMALIZED.getInstance());

    EVENTS.put("PERFORMANCEORMFAILED",marf.net.assl.generatedbyassl.as.dmarf.events.PERFORMANCEORMFAILED.getInstance());

    AES.put("STAGE_AE", marf.net.assl.generatedbyassl.as.aes.STAGE_AE.getInstance());

    static public DMARF getInstance ( )
    {
        if ( null == oInstance )
        {
            oInstance = new DMARF();
        }
        return oInstance;
    }

    public static void main ( String[] args )
    {
        DMARF oANTS = DMARF.getInstance();
    }
}

```

Fig. 24. DMARF's Singleton Pattern Code

allow a global point of access to the object, and allow multiple instances in the future without affecting a singleton class's clients[22].

Singleton pattern is needed in DMARF to provide a global visibility or a single access point to a single instance of the class DMARF, in this case, always a unique instance of DMARF class will be instantiated to avoid the synchronization problems that can be raised. The constructor is assigned

to be private, in order to prevent direct instantiations from other clients, and to define other subsystems based on this extending . The static method getInstance (), is to ensure that a single instance of DMARF object is returned, it allows the instantiation of DMARF elements just when the singleton instance is equal to null. Figure 23 , shows a UML class diagram for the singleton Pattern in DMARF. Figure 24, shows source code for singleton pattern in DMARF.

```
public interface IDemandGenerator
{
    void setDemandDispatcher(IDemandDispatcher poDispatcher);
    void setGEER(GIPSYProgram poGEER);
    String generateDemand(int piID, int[] paiContext);
}

public class DemandWorker implements IDemandWorker
{
    protected ITransportAgent oTA;
    protected LocalDemandStore oLocalDemandStore;
    protected LocalGEERPool oLocalGEERPool;
    protected volatile boolean bIsWorking = true;
    private TAEExceptionHandler oTAEExceptionHandler = null;
    private String strWorkerID = null;

    public DemandWorker()
    {
    }
    public void startWorker()
    {
    }
    public void stopWorker()
    {
    }
    public void setTransport()
    {
    }
}

Class LocalDemandStore
{
    public boolean contain() { ... }
    public void put () {... }
    public update()
    {
    }
}

Interface IDemand
{
    public void getSignature() {
    ...
    }
    public void setDemand() {
    ...
    }
    public DemandState getDemand() {
    ...
    }
    public Demand AttachDemand(LocalDemandStore) {
    ...
    }
    public Demand detachDemand(LocalDemandStore) {
    ...
    }
}
```

Fig. 26. GIPSY's Observer Pattern Code First Part

2) GIPSY:

Observer Pattern:

Observer pattern is known as publish subscribe. Define a one to many dependency between object so that when one

Interface ITransportAgent

```
{
    public string getDemand() {
    ...
    }
    public IDemand getDemand() {
    ...
    }
    public concreteobserver getState() {
    ...
    }
    public void setState(concreteobserver) {
    ...
    }
}
```

Fig. 27. GIPSY's Observer Pattern Code Second Part

```
public class GIPC
extends IntensionalCompiler
{
    private Preprocessor oPreprocessor = null;
    private Dictionary oDictionary = null;
    private GEERGenerator oGEERGenerator = null;
    private IdentifierContextCodeGenerator oICCodeGenerator = null;
    private Translator oTranslator = null;
    private GIPSYProgram oGIPSYProgram = null;
    private AbstractSyntaxTree oPreprocessortAST = null;

    public Dictionary getDictionary() {
        return this.oDictionary;
    }

    public AbstractSyntaxTree compile(Object poExtraArgs) throws GIPCException {
        init();
        process();
        return this.oAST;
    }

    public GIPSYProgram getGIPSYProgram(){
        return this.oGIPSYProgram;
    }

    public GIPSYProgram getGEER(){
        return getGIPSYProgram();
    }

    public AbstractSyntaxTree translate() throws IntensionalCompilerException {
        for(int i = 0; i < this.aoIntensionalCompilers.length; i++)
        {
            AbstractSyntaxTree oCurrentIntensionalAST =
                this.aoIntensionalCompilers[i].translate();
        }
        return this.oAST;
    }
}
```

Fig. 29. GIPSY's Facade Pattern Code

object changes state, all its dependents are notified and updated automatically. The need to maintain consistency between related objects is a common side-effect of partitioning a system into a collection of cooperating classes [22].

Using the observer pattern could be suitable once: an abstraction has two aspects one dependent on the other; a change to one object requires changing others and you do not know how many objects need to be changed; or an object should be able to notify other objects without making assumptions about who these objects are.

Observer pattern has couple participants assisting and cooperating to each other in order to achieve their target. Subject knows its observers, and provides an attaching and detaching

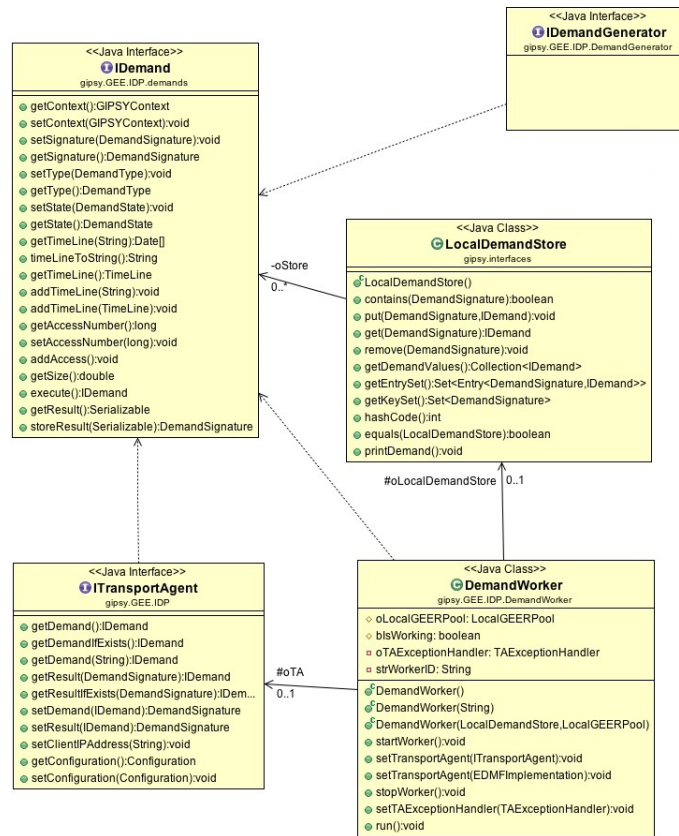


Fig. 25. GIPSY's classes involved in the Observer pattern.

interface observer object. Observer defines an updating interface for object that should be notified. Concrete subject stores state of interest to concrete subject object. Concrete observer maintains a reference to concrete subject object.

GEE's observer pattern separates the presentation of user interface from underlying application data is one of aspect which could be reused independently. Once Demand generator (a subject) gets executed and changed its state, LocalDemandStore (observer), contains demand signature, will notify Demand Worker (concrete observer) his new state; thereby TransportAgent (concrete subject) will returns subject state. This behavior implies that Demand generator and RransportAgen are dependent on the data object, and get notified of any changes in its state. Indeed, observer pattern describes how to establish these relationship. All observers are informed whenever the subject undergoes a change in state and synchronize its state as well. Figure 25, shows a UML class diagram for the observer pattern in DMARF. Figure 26 and 27, shows source code for observer pattern in GIPSY.

Facade Pattern:

The facade pattern provides an interface for a large amount of code and makes it easier to use the code library. It reduces the dependencies outside the code library and hides the implementation of the subsystems from any clients and makes it easy to use. It also makes the code more easier to read and

understand. So this structuring reduces the complexity of the subsystem. These subsystems can be any groups of classes. [22]

Figure, facade is interacting with the different classes and GIPC acts as an interface for all these classes. GIPC calls the methods from the classes like GIPSYProgram, Preprocessor, GEERGeneration, Translator within the main class. So the developer could easily interact with the methods of the other classes without interfering with the underlying code. It can also call the other class methods and attributes using the facade. Figure 28, shows a UML class diagram for the facade Pattern in GIPSY. Figure 29, shows source code for facade pattern in GIPSY.

Strategy Pattern:

The strategy pattern define sets of algorithms where each set is encapsulated in order to be interchangeable. This pattern allows algorithms to differ independently from clients that use it. More importantly, this pattern retains the open/closed and reuse principle [25].

In GIPSY this pattern is applied where there are two implementation options can be set to the TransportAgent, Jini and JMS. All TAs are implementing the ITransportAgent interface; the implementations of the two DMSs, Jini and JMS are encapsulated in IJMSTransportAgent and IJiniTransportAgent which in turn implement the ITransportAgent hence they


```

public abstract class GenericTierWrapper implements IMultiTierWrapper
{
    protected IDemandDispatcher oDemandDispatcher = null;

    public void setConfiguration(Configuration poConfiguration)
    {
        this.oConfiguration = poConfiguration;
    }

    public void setTransportAgent(ITransportAgent poTA)
    {
        this.oDemandDispatcher.setTA(poTA);
    }

    public void setDemandDispatcher(IDemandDispatcher poDemandDispatcher)
    {
        this.oDemandDispatcher = poDemandDispatcher;
    }

    public IDemandDispatcher getDemandDispatcher()
    {
        return this.oDemandDispatcher;
    }
}

public abstract class DemandDispatcher implements IDemandDispatcher
{
    protected ITransportAgent oTA = null;

    public DemandDispatcher()
    {
    }

    public void setTA(ITransportAgent poTA)
    {
        this.oTA = poTA;
    }

    public ITransportAgent getTA()
    {
        return this.oTA;
    }
}

```

Fig. 31. GIPSY's Strategy Pattern Code

- [3] Serguei A. Mokhov, Lee Wei Huynh, and Jian Li. Managing distributed MARF with SNMP. Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada, April 2007. Project report. Hosted at <http://marf.sf.net> and <http://arxiv.org/abs/0906.0065>, last viewed February 2011.
- [4] Serguei A. Mokhov and Rajagopalan Jayakumar. Distributed Modular Audio Recognition Framework (DMARF) and its applications over web services. In Tarek Sobh, Khaled Elleithy, and Ausif Mahmood, editors, Proceedings of TeNe'08, pages 417–422, University of Bridgeport, CT, USA, December 2008. Springer. ISBN 978-90-481-3661-2. doi: 10.1007/978-90-481-3662-9_72. Printed in January 2010.
- [5] Serguei A. Mokhov, Emil Vashev, Joey Paquet, and Mourad Debbabi. Towards a self-forensics property in the ASSL toolset. In Proceedings of the Third C* Conference on Computer Science and Software Engineering (C3S2E'10), pages 108–113, New York, NY, USA, May 2010. ACM. ISBN 978-1-60558-901-5. doi: 10.1145/1822327.1822342.
- [6] Serguei A. Mokhov. Towards security hardening of scientific distributed demand-driven and pipelined computing systems. In Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPD'08), pages 375–382. IEEE Computer Society, July 2008. ISBN 978-0-7695-3472-5. doi: 10.1109/ISPD.2008.52.
- [7] Yi Ji, Serguei A. Mokhov, and Joey Paquet. Unifying and refactoring DMF to support concurrent Jini and JMS DMS in GIPSY. In Bipin C. Desai, Sudhir P. Mudur, and Emil I. Vashev, editors, Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering (C3S2E'12), pages 36–44, New York, NY, USA, June 2010–2013. ACM. ISBN 978-1-4503-1084-0. doi: 10.1145/2347583.2347588. Online e-print <http://arxiv.org/abs/1012.2860>.
- [8] Joey Paquet and Ai Hua Wu. GIPSY - a platform for the investigation on intensional programming languages. In Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005), pages 8–14. CSREA Press, June 2005. ISBN 1-932415-75-0.
- [9] Joey Paquet and Peter G. Kropf. The GIPSY architecture. In Peter G. Kropf, Gilbert Babin, John Plaice, and Herwig Unger, editors, Proceedings of Distributed Computing on the Web, volume 1830 of Lecture Notes in Computer Science, pages 144–153. Springer Berlin Heidelberg, 2000. doi: 10.1007/3-540-45111-0_17.
- [10] Bin Han, Serguei A. Mokhov, and Joey Paquet. Advances in the design and implementation of a multi-tier architecture in the GIPSY environment with Java. In Proceedings of the 8th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2010), pages 259–266. IEEE Computer Society, May 2010. ISBN 978-0-7695-4075-7. doi: 10.1109/SERA.2010.40. Online at <http://arxiv.org/abs/0906.4837>.
- [11] Serguei A. Mokhov and Joey Paquet. Using the General Intensional Programming System (GIPSY) for evaluation of higher-order intensional logic (HOIL) expressions. In Proceedings of the 8th IEEE / ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2010), pages 101–109. IEEE Computer Society, May 2010. ISBN 978-0-7695-4075-7. doi: 10.1109/SERA.2010.23. Pre-print at <http://arxiv.org/abs/0906.3911>.
- [12] Serguei A. Mokhov, Emil Vashev, Joey Paquet, and Mourad Debbabi. Towards a self-forensics property in the ASSL toolset. In Proceedings of the Third C* Conference on Computer Science and Software Engineering (C3S2E'10), pages 108–113, New York, NY, USA, May 2010. ACM. ISBN 978-1-60558-901-5. doi: 10.1145/1822327.1822342.
- [13] SonarSource. SonarQube, 2014. <http://www.sonarqube.org/downloads>.
- [14] Documentation - SonarQube - Codehaus. SonarSource [Online]. Available: <http://docs.codehaus.org/display/SONAR/Documentation>. [Accessed: Jul 22, 2014].
- [15] Codehaus - SonarQube [Online]. Available: <https://jira.codehaus.org/browse/SONAR>. [Accessed: Jul 22, 2014]
- [16] Serguei A. Mokhov, Miao Song, and Ching Y. Suen. Writer Identification Using Inexpensive Signal Processing Techniques. [online], 30 Dec 2009, <http://arxiv.org/abs/0912.5502>.
- [17] Marius Bulacu, and Lambert Schomaker. Text-Independent Writer Identification and Verification Using Textural and Allographic Features. IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE. Published by the IEEE Computer Society, VOL. 29, NO. 4, APRIL 2007.
- [18] Emil Vashev and Serguei A. Mokhov. Towards autonomic specification of Distributed MARF with ASSL: Self-healing. In Proceedings of SERA 2010 (selected papers), volume 296 of SCI, pages 1–15. Springer, 2010. ISBN 978-3-642-13272-8. doi: 10.1007/978-3-642-13273-5_1.
- [19] Mokhov, Serguei A. (2005) Towards Hybrid Intensional Programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY. Masters thesis, Concordia University.
- [20] Mokhov, Serguei A. and Paquet, Joey and Debbabi, Mourad (2011) Reasoning About a Simulated Printer Case Investigation with Forensic Lucid. In: International ICST Conference on Digital Forensics and Cyber Crime (ICDF2C), October 2011, Dublin, Ireland.
- [21] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. "Distributed Systems Concepts and Design", Fifth Edition, Addison-Wesley, 2012, ISBN: 0-13-214301-1
- [22] Craig Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Pearson Education, third edition, April 2006. ISBN: 0131489062.
- [23] A. Shalloway, J. Trott, Design Patterns Explained, Addison-Wesley, NJ, June 2001, ISBN-10: 0321247140.
- [24] Pankaj. (July 2, 2013). Adapter Design Pattern in Java: Example Tutorial. Available: <http://www.journaldev.com/1487/adapter-design-pattern-in-java-example-tutorial>. Last accessed Aug 11, 2014.
- [25] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. Head First Design Patterns. O'Reilly and Associates, Inc., first edition, October 2004.
- [26] Han, Bin and Mokhov, Serguei A. and Paquet, Joey (2009) Advances in the Design and Implementation of a Multi-tier Architecture in the

- GIPSY Environment with Java. In: Software Engineering Research and Applications (SERA 2010), Montreal, QC, Canada.
- [27] ObjectAid UML Explorer for Eclipse. Available: <http://www.objectaid.com/>. Last accessed Aug 11, 2014.
- [28] Patkos Csaba. (12 Mar 2013). Techniques for Refactoring Code. Available: <http://code.tutsplus.com/courses/techniques-for-refactoring-code>. Last accessed Aug 27, 2014.

TABLE II
PAPER DISTRIBUTION: FIRST DELIVERABLE

Name		Paper Title
Abdulrhman Albeladi	DMARF	[2]On design and implementation of distributed modular audio recognition framework: Requirements and specification design document.
	GIPSY	[11] Using the General Intentional Programming System (GIPSY) for evaluation of higher-order intentional logic (HOIL) expressions.
Aber Abozkhar	DMARF	[3]Managing distributed MARF with SNMP
	GIPSY	[9]The GIPSY architecture
Ahmed Almessabi	DMARF	[5]Towards a self-forensics property in the ASSL toolset
	GIPSY	[8]GIPSY a platform for the investigation on intentional programming languages.
Huda Mohamed	DMARF	[1]Autonomic specification of self-protection for Distributed MARF with ASSL
	GIPSY	[7]Unifying and refactoring DMF to support concurrent Jini and JMS DMS in GIPSY.
Jilson Thomas	DMARF	[6]Towards security hardening of scientific distributed demand-driven and pipelined computing systems
	GIPSY	[10]Advances in the design and implementation of a multi-tier architecture in the GIPSY environment with Java.
Zakaria Alomari	DMARF	[4]Distributed Modular Audio Recognition Framework (DMARF) and its applications over web services.
	GIPSY	[12]Towards a self-forensics property in the ASSL toolset.

TABLE III
DESIGN PATTERN DISTRIBUTION: THIRD DELIVERABLE

Name	Design Pattern
Abdulrhman Albeladi	Factory
Aber Abozkhar	Singleton
Ahmed Almessabi	Observer
Huda Mohamed	Strategy
Jilson Thomas	Facade
Zakaria Alomari	Adapter

TABLE IV
TERMINOLOGY

Term	Definition
MARF	Modular Audio Recognition Framework
DMARF	Distributed MARF
NLP	Natural Language Processing
CORBA	Common Object Request Broker Architecture
XML-RPC	Remote Procedure Call (RPC) protocol which uses XML to encode its calls
Java RMI	Java Remote Method Invocation
API	Application Programmers Interface
PDA	personal digital assistant
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
RMI	Remote Method Invocation
WS	Web Services
ASSL	Autonomic System Specification Language
GIPSY	General Intensional Programming System
AS	Autonomic System
AE	Autonomic Elements
ASIP	AS Interaction Protocol
GIPC	General Intensional Programming Compiler
GEE	General Education Engine
JVM	The Java Virtual Machine.
RPC	Remote Procedure Call
WSDL	Web Services Definition Language